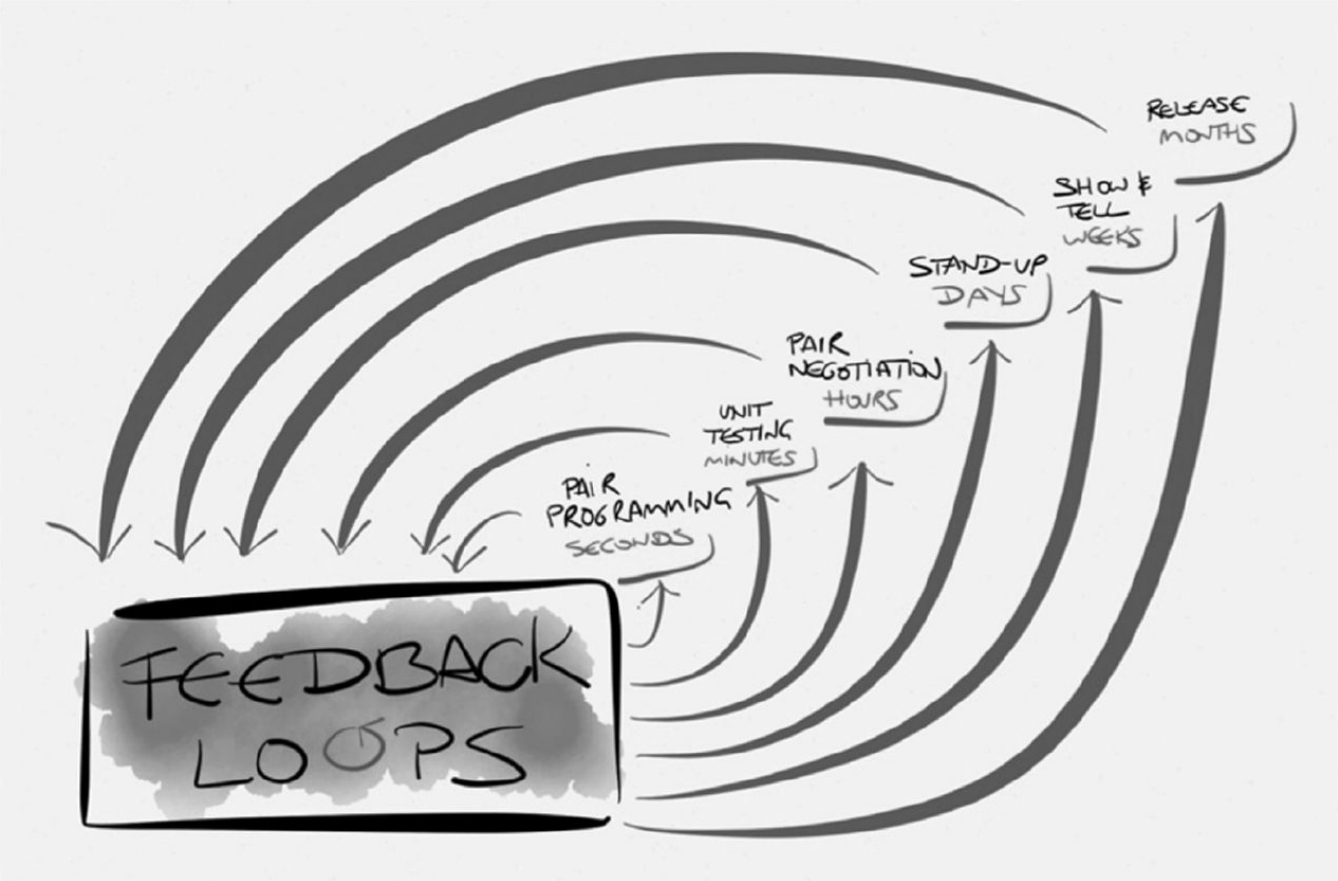
8 COMMON AGILE PRACTICES

In this Chapter we detail the main Agile practices that are generic across most or all frameworks.

8.1 SHORT FEEDBACK LOOPS

Feedback loops are critical to the success of an Agile delivery. An empirical process is one where a team inspect how and what they have done, and use this feedback to improve their process and products – this is called a feedback loop (see Figure 8.1).

Figure 8.1 Feedback loops



The following are some examples of feedback loops:

- Face-to-face conversations (see Section 8.2).
- Daily stand-ups (see Section 8.3).
- Show and tells (see Section 8.4).

Other forms of feedback loops, such as pair programming (Section 14.1), unit testing and continuous integration (Section 8.10) and regular releases (Section 10.2) are covered later in the book.

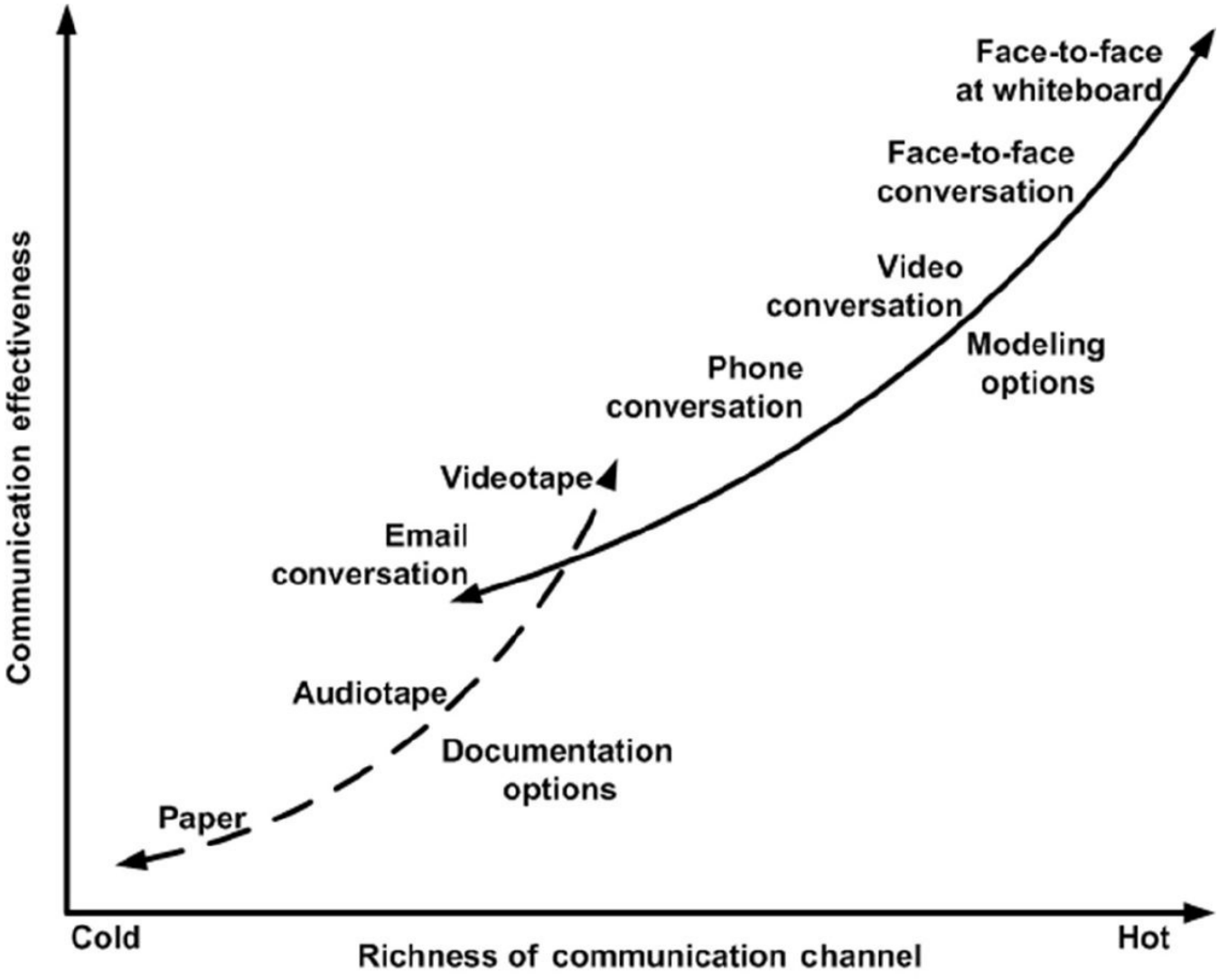
8.2 FACE-TO-FACE COMMUNICATION

Face-to-face communication enables fast feedback loops and adds rich non-verbal

communication to any interaction. It also solidifies relationships, creates mutual trust and defines centres of influence within an organisation.

Figure 8.2 shows the relationship between the communication channel being used and the effectiveness and richness of communication (Ambler, 2001–14)

Figure 8.2 Richness of communication channels



In collaborative work, face-to-face interaction is a more favourable choice, in particular if a relatively more difficult and significant task is involved – discovery sessions, kick-off meetings and contract negotiations rarely take place via mediated communication such as audio, video, email and so on.

In Agile deliveries, it is recommended that teams are physically co-located to enable face-to-face communications and the benefits it brings. In the next sections we will discuss why face-to-face communications are key to enable Agile delivery and how to facilitate it for distributed teams.

Number of communication channels

The more people are involved in a communication, the more difficult it becomes to communicate effectively. To highlight this communication challenge, Kerzner

(2013) looks at how the number of communication lines (channels) within a defined group of people affects communication. As shown in the table below, the number of communication channels increases disproportionally to the number of people in a group.

People	2	3	4	5	6	7	8	9	10
Channels	1	3	6	10	15	21	28	36	45

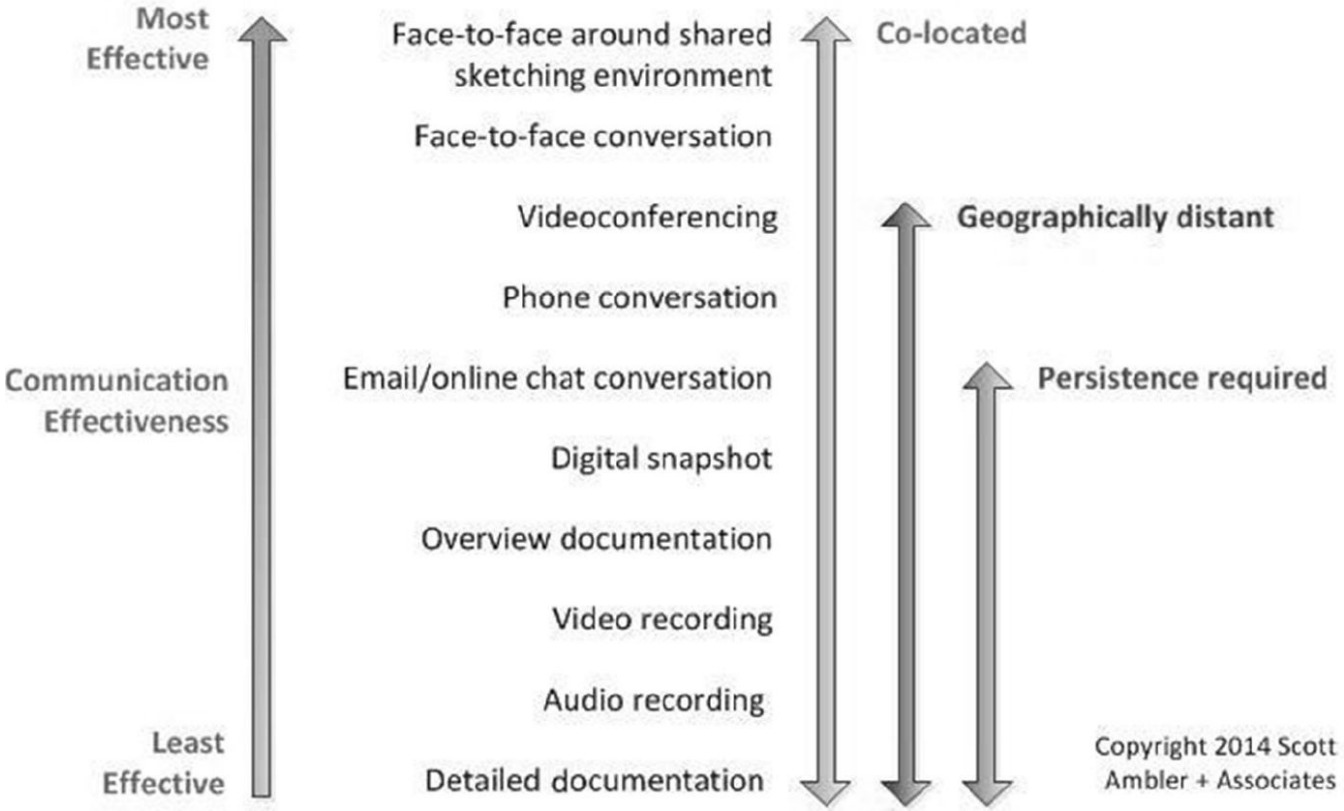
This is why Agile teams tend to consist of between 3 (to enable team dynamics) and 11 members (to restrict communication channels to a manageable size face-to-face).

Team proximity

In spite of promising technology achievements, such as high-resolution video conferencing at affordable cost, project management platforms and mobile phone applications, proximity still tends to be the preferred viable option for effective collaboration.

Figure 8.3 shows the impact of proximity on effective communication (Ambler, 2001–14). Proximity generates spontaneous, memorable and sustainable interactions, as it provides numerous unscheduled opportunities that can lead to conversation: in designated recreation areas, in the office kitchen or in lifts. Such interactions, known as ‘water cooler conversations’, have powerful effects on decision making, conformity, social pressure and familiarity. The ‘mere exposure effect’ (Zajonc, 1968) can facilitate problem solving, product development and task coordination.

Figure 8.3 Effects of proximity



The frequency of spontaneous communication is predominantly correlated with office distance, with frequency dropping exponentially as the distance between the two collaborators increases.

> In early collaborative studies of competitive games (Deutsch, 1958), participants were strongly advised to 'win as much as they could for themselves' when competing against their opponent. Observations showed that 71 per cent of participating 'couples' made cooperative choices for a common cause in cases where communication was allowed. In trials during which communication was strictly prohibited, only 36 per cent of participants fought for a common goal. Evidently, such behaviour derives from the silent commitment imposed by face-to-face social contracts, in addition to the group identity that is formed through face-to-face interactions.

Osmotic communication

This term was introduced by Alistair Cockburn (Cockburn, 2004) and refers to the boundaryless information flows amongst co-located team members as part of their daily conversations and interactions. For instance, two colleagues engage in conversation about exploring a solution to a complex problem. Another team member overhears the discussion and, in order to provide her colleagues with an alternative solution – which could potentially be the solution that they are looking for – she decides to join in the conversation. Such interaction would not be possible in distant collaboration.

Tacit knowledge

Tacit knowledge is knowledge that cannot be codified, is difficult to communicate in written form and is normally communicated face-to-face. Tacit knowledge can be individual tacit knowledge or group tacit knowledge.

At the individual level, tacit knowledge is closely related to the concept of accumulative knowledge, based on a plethora of real-life events. It is knowledge that is derived from the stock of learning activities and is expressed in public through skills. At the group level, team knowledge is based on common experiences.

8.2.1 Distributed team communication challenges

There are two main types of non-physically located teams:

- **Multisite team** This refers to one product group that is split up across various locations into smaller teams.
- **Distributed team** This means that individual team members are located at different sites.

It is worth noting that geographic dispersion is not limited to different offices, countries or continents – it can happen in the same office, within rooms (with space dividers), between different rooms (with physical walls) or between floors.

The primary economic driver to embrace distributed or multisite teams, in

particular in the context of offshoring, is to minimise operational cost. Beyond apparent financial gains, organisations benefit from offshoring due to new innovation, speed, agility and new revenue opportunities (Carmel and Tija, 2005). However, this geographic dispersion brings with it the communication challenges of distance and time, which cannot be completely eliminated.

8.2.1.1 Five centrifugal forces of distributed teams

The virtual distance amongst knowledge workers can be further analysed by looking at the following five centrifugal forces of distributed teams (Carmel and Tija, 2005).

Force 1: Communication breakdown In any face-to-face communication, humans interpret body language (kinesics), voice (paralanguage), touch (haptics), distance (proxemics) and environmental characteristics. It is believed that 80 per cent of exchanged messages are non-verbal, including voice quality, rate, pitch, volume, accent, blink rate, hand gestures, eye contact, mouth gestures and so on. It is more difficult to enable effective communication with geographically dispersed teams because none of these elements are easily put in place.

Force 2: Coordination breakdown Agile development relies on frequent inspection and adaptation. Through face-to-face interactions, teams make decisions on adjustments to realign the product increment with the overall objective in the following iteration/sprint. These decisions are often based on numerous ad-hoc conversations around small adjustments: brief discussions about design and architecture, requests for clarification on user experience, questions about the product vision and so forth. If team members are located in close proximity, such conversations can occur spontaneously; if this proximity is missing then there is a significant risk of coordination breakdown.

Force 3: Loss of communication richness Distributed environments tend to lack communication richness. This introduces various challenges and negatively influences the success of interactions: messages might only be partially transmitted, acknowledged (in absence of non-verbal cues) and partially understood (for example, due to being unable to immediately interact with the transmitter and ask for clarification). Messages with missing information often lead to actions that are misaligned with the goal, delays (for example, due to asynchronous receipt of messages; time zone differences; need for clarifications), rework (for example, due to lack of deep understanding) and conflict (because of the emotional involvement of the team members).

Force 4: Loss of team bonding Face-to-face communication is a vital element in team bonding and establishing trust among team members. Trust, in turn, is a significant contributor to team productivity and job satisfaction. In distributed environments, it is often challenging to establish trust amongst the team unless some prior working relationships exist. This leads to distributed teams lacking cohesion and, in the worst case, to mistrust, which will have a disastrous effect on teams.

Force 5: Cultural differences In particular when offshoring it is likely that teams

will be made up of people from different cultures and backgrounds. Lack of cultural understanding is often a factor that introduces conflict and mistrust, which will lead to project failure.

According to Hofstede (1994), cross-cultural communication is prone to misinterpretation, as transmitter and receiver comprehend the same verbal and non-verbal message differently. Face-to-face communication builds an environment of trust, and is therefore less likely to be misinterpreted and more likely to foster cultural understanding.

The challenges listed above can be further exacerbated if utilising more traditional delivery approaches that rely on documentation for communication. When communication is already a significant issue, then just passing very detailed documents between team members runs the risk of making communication issues significantly worse, mainly because of the risk of misinterpretation.

8.2.2 Distributed team communication risk mitigation

Due to the issues discussed above, replicating the benefits of face-to-face communication in distributed teams presents numerous challenges. The following virtual co-location tools can help to mitigate the impact:

- **Video conferencing** Building rapport and eliminating barriers between teams, such as the subtle us–them attitude, is achievable through video communications as it allows members to pick up on non-verbal cues. Video conferencing is ideal for daily stand-up meetings, sprint reviews and retrospectives. For distributed teams it is recommended to engage in video sessions as frequently as possible.
- **Knowledge management systems** Creating a knowledge cube (see Chapter 5) of organisational knowledge that contains data from the broader spectrum of the business, be it finance, marketing, product, software engineering and so on, can be particularly beneficial for productivity and boundaryless information flow. It provides a go-to place for continuous reference to explicit knowledge.
- **Collaboration platforms** Distributed development and everyday collaboration is frequently reliant on the extensive use of collaboration platforms. A unified communication approach involves task-oriented communication, with comments, discussion boards and file transfer capabilities related to specific work items. It also gives Agile leads, customers, the team and other stakeholders better visibility of the status of work, with transparency of all communications.
- **Instant messaging (IM)** This tool can help to replicate spontaneous communication up to a certain extent. It is particularly useful for short questions and requests for brief clarifications. Instant messaging virtually brings co-workers closer, though it cannot fully replace the benefits of face-to-face communication.
- **Interactive whiteboards** Interactive whiteboards can prove useful especially

in design and architecture sessions that have to take place remotely. Whiteboard content can be shared in real-time across multiple locations, allowing distributed participants to collaborate using inclusive techniques, such as drawings, diagrams, low-fidelity user interface wireframes, and other forms of Agile modelling.

8.2.2.1 Cross-pollination

Cross-pollination is based on mobilising resources from one development site to the other. Especially during project kick-off activities, it is important to create a platform for face-to-face group workshops, pair programming sessions and so on. Resources should not be limited to the management level because communication needs to happen across the whole team. All team members should be an essential part of cross-pollination, allowing them to interact with business and decision-making stakeholders.

8.3 DAILY STAND-UPS

Daily stand-ups (or daily synchronisation meetings) are, as the name suggests, daily meetings that allow a team to inspect the progress that has been achieved, and then plan out the day's work, including any corrective actions needed to ensure that the best results obtainable during the iteration are delivered.

The meetings should be limited to a maximum of 15 minutes. One way to achieve this is for the attendees to remain standing (which is why it is called the 'daily stand-up'). If an issue arises that requires more discussion, then it should be discussed by the affected team members in a separate meeting, which should be held immediately afterwards.

Daily stand-ups are not a project management or progress update meeting; instead the focus is on synchronisation within the team. Usually these meetings follow a very simple format, with each team member answering each one of the following three questions:

- What did I do yesterday that helped the team meet the sprint/iteration goal?
- What will I do today to help the team meet the sprint/iteration goal?
- Do I see any impediment that prevents me or the team from meeting the sprint/iteration goal?

An alternative to this approach, which works well for teams who are focused on keeping a continuous flow of work that is balanced across everyone in the team, is to 'walk the board' – a reference to the visual board around which these meetings are normally centred (see Section 8.7).

In this approach (assuming tasks flow from left to right on the visual board), a team would start from the rightmost point on the board and talk about the tasks (and/or stories) that are nearest completion. Then they work their way back across the board. This method is based on the assumption that it is most important to complete the tasks that are nearest to completion as they would be the highest

value ones.

The best time of day to hold a daily stand-up is about 30 minutes into the working day. This allows team members to arrive at their desks and refresh their memory of the previous day before attending the meeting. The daily stand-up will then give them their priority work for the day.

Of course, this is an ideal that won't suit every team – for example, if some team members work in different locations or even different countries, then the timing of the meeting should be set so everyone can attend. Including remote teams or workers in a stand-up meeting can prove problematic as not everyone might be able to see or have access to the visual board, which provides a common reference point. Fortunately software tools can help manage the board so all team members can see it and video streaming technology can allow all team members to see each other and a physical board at the meeting.

8.4 SHOW AND TELLS

'Show and tells' are meetings that are typically held at the end of a sprint/iteration and /
or release (see Section 5.1). They are an essential part of the Agile inspect-and-adapt cycle.

The purpose of these meetings is for the team to demonstrate to the stakeholders all stories completed during an iteration/sprint, to seek immediate feedback from the stakeholders, as well as get recognition for their work. This means that all available stakeholders and all team members should attend show and tells.

Show and tells should be live demonstrations of a new working product or feature as it is essential that stakeholders get to see, touch and feel the real product. A show and tell that uses screenshots, simulations or models of the product will not add as much value as a demonstration of real working software; only by seeing the real working product can the stakeholders be able to provide useful and accurate feedback.

The person who demonstrates a product depends on each team's circumstances. For example, more experienced team members or those people who are comfortable speaking in front of an audience may present. Some teams may pick the same person to run the demonstration every iteration/sprint for a consistent approach. Other teams may rotate the job to ensure everyone has an opportunity to run the demonstration should they so wish.

Alternatively, if a team has a very engaged customer, having the customer do the demonstration adds a lot of gravitas as it is essentially a presentation 'from business person to business person'. In all circumstances it is bad practice to only show completed stories to the customer at a show and tell, because if the customer starts highlighting that the stories are incorrect at this late point it sends a very negative message to other stakeholders who are in attendance.

Show and tell meetings should also provide the opportunity to identify stories that were planned but not completed within the iteration/sprint, and to agree a date

when they will be achieved. They can also be a good time to highlight risks, issues, blockers and assumptions the team are working with as stakeholders may offer good insight and information that may assist with the resolution of these. Any defects of the product that have been identified should also be shared.

> Teams should avoid presenting stories that are only nearly complete at ‘show and tell’ meetings as this can give stakeholders the false impression that stories are more advanced in their development than they truly are.

Some up-front preparation is needed for show and tell meetings. For example the product needs to be doubled-checked to ensure all its new stories are working in the environment where the demonstration will take place. The team also need to prepare the location for the show and tell and check the facilities such as network communications. It can also be useful to rehearse the demonstration in the place where it will happen. After the meeting any output should be fed into the backlog and prioritised potentially for development during the next time-box.

8.5 RETROSPECTIVES

Traditionally, project management has embraced the concept of lessons learnt and post-mortem meetings, which capture the learning outcomes of a project, post completion or termination. Such lessons would be shared with the rest of the organisation to consider in future projects that exhibit common characteristics, for example, similar business requirements, technical domain, team structures, collaboration dynamics.

Retrospectives provide opportunities for the team and others to specifically focus on inspecting and adapting what they do and how they do it. The purpose of retrospectives is to continually improve the product as it is being delivered and to continuously improve how the products are delivered. Teams should analyse how well they performed by identifying things that have worked well and those that could be improved. Retrospectives are a key element to fulfil the continuous improvement mentality.

Retrospectives are most commonly scheduled at the end of each sprint/iteration, although they should not be limited to this – they should also be held at any other appropriate point if required, for example, after a release has been completed or following issues that may have occurred.

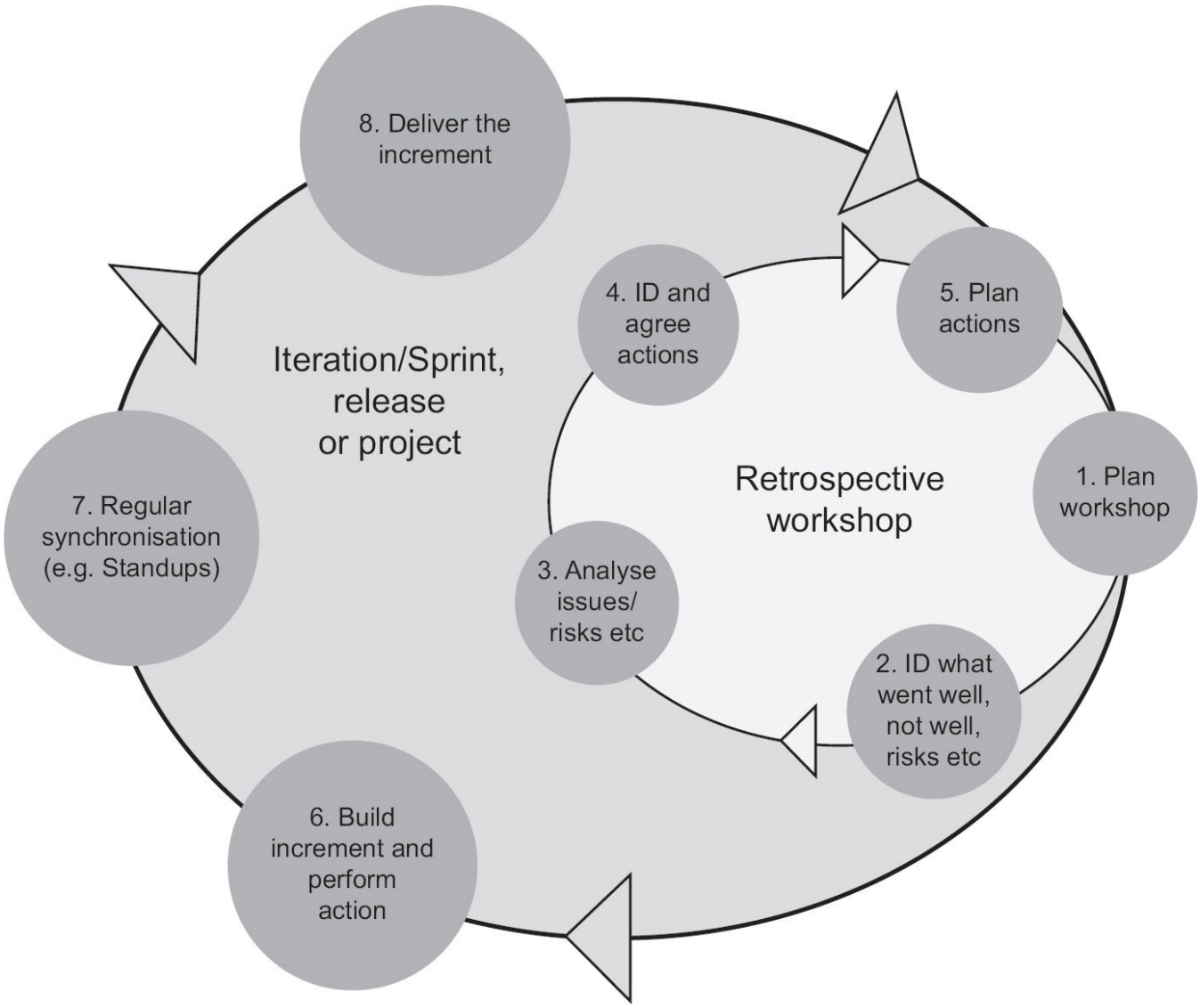
There are many different types of retrospectives, depending on what the team are reflecting upon and adjusting, and there are many different techniques that can be used. For example, at the end of the sprint/iteration time-box the team will typically spend a few hours considering:

- What went well?
- What didn’t go well?
- What are we going to do differently next time?

Figure 8.4 shows a typical retrospective cycle, from the delivery of an

iteration/sprint through to the retrospective.

Figure 8.4 Agile retrospective process



- **Plan workshop** Give some background about why the retrospective is being performed, how the workshop objectives are going to be met and what it is the team are going to do. This may be described in a workshop agenda that is sent out prior to the retrospective workshop.
- **Identify what went well, what didn't go well, risks and so on** There are numerous ways to do this, and usually using some type of 'brainstorming' technique works well. The focus here is on understanding what will be discussed in this retrospective workshop. It is important to keep in mind that it is as important to celebrate success (what went well) as it is to understand what didn't go well.
- **Analyse issues** There are numerous ways to generate insights depending on what the retrospective is focused on. A few retrospective activities will be discussed in Section 8.5.1. Also review issues and actions that have been raised in previous retrospectives.
- **Identify and agree actions** Identify actions as required and agree within the team that the focus within the next iteration/sprint, release or project will be on these actions.
- **Build increment and perform actions** Build the next increment of the product and deliver the actions that were agreed upon in the last retrospective.

- **Regular synchronisation** Synchronise regularly to ensure that the actions are on track.
- **Deliver the increment** Deliver the product increment from the iteration, including any actions agreed from the last retrospective.
- **Deliver the cycle again** Kaizen or continuous improvement.

The insights gathered in a retrospective can be of use to the existing project, business domain, technical domain and team, and the added-value knowledge can be applied in subsequent iterations/sprints.

Retrospectives generate a number of action points, which should contribute to and improve the following aspects:

- collaboration;
- productivity;
- quality;
- capability;
- capacity;
- team dynamics.

The team should also reflect on how they work together and how they interact with the rest of the business as a team. Regarding their productivity, the team should identify opportunities to reduce rework to allow for more productive work to be done.

Quality should be a constant focus throughout an Agile delivery. Therefore, the team should assess their approach to delivering better code with fewer defects. Also, the team should focus on improving their capacity by identifying efficiency improvements. Finally, close collaboration may introduce friction and conflict, and retrospectives are an appropriate time to express concerns and emotions and discuss possible resolutions.

Facilitating retrospectives

It is usually the Agile lead (see Section 6.3) who acts as a facilitator in a retrospective and encourages team members to commit to a list of action points. During retrospectives, it is important to make sure that the team identify things that are within their sphere of influence, rather than concentrating on the ones that will change the world or reinvent the wheel.

There is also the danger that team members may air all of their more petty grumbles (like ‘the coffee isn’t nice enough’). These grumbles might be acceptable the first time they are raised and can be passed onto the person responsible; however, it is important to move on from this and help the team to identify things that improve their process, tools, culture, ways of working and so on.

If retrospectives are run in the same format all the time every time, teams might become bored, which in turn will make the meeting ineffective. Some measures to

counteract this are to change location or to vary the formality of the meetings.

8.5.1 Examples of retrospective activities

8.5.1.1 Explorers/Shoppers/Vacationers/Prisoners

Over time, team engagement towards the retrospectives may diminish for a variety of reasons. To discuss the underlying reasons behind low engagement, a facilitator asks the team members to associate themselves with one of the following behaviours:

- **Explorers:** eager to discover new ideas and insights, with desire to gather all available information about the iteration/release/project;
- **Shoppers:** willing to expose themselves to available information and select one useful new idea;
- **Vacationers:** no interest in the works of the retrospective, but happy to be away from their desk;
- **Prisoners:** feel forced to attend and would prefer to be elsewhere.

The facilitator can then assess if the style or focus of the retrospective needs to change.

8.5.1.2 Timeline

At times, teams can be called to identify the root cause of an issue and its progression before taking any action. In the timeline technique, a facilitator draws a timeline on a whiteboard, which represents the period under review. The participants are asked to write events on coloured sticky notes, in the form of good, problematic, and significant. Following that, they are called to mount them on the whiteboard in relation to the timeline. The exercise provides the team with better insights into the root cause of problems, allowing them to reflect and understand how to avoid similar situations in a future iteration.

8.5.1.3 Five whys

The five whys exercise deals with causation, i.e., the relationship between an event (cause) and another event (effect). It helps identify the true root cause of a problem by asking why repetitively and bypassing evident answers that do not shed light on the investigation. Here is an example:

> Event: The team can never focus on delivery and is therefore failing to deliver iteration/sprint goals.
>
> Why: Management keep interrupting the team.
>
> Why: Management keep asking for progress updates.
>
> Why: They can’t see the daily status of the iteration/sprint and don’t know if the team are on track.
>
> Why: The burn-down chart is always out of date and doesn’t show the accurate delivery status.

Why: The burn-down chart isn’t being updated daily by the team.

8.5.1.4 Fishbone diagram

A fishbone diagram provides a visual approach to help with root cause analysis of problems, showing them alongside their contributing factors. A facilitator draws a simplified fishbone diagram on a whiteboard, placing the problem at the head. In a collaborative manner, the participants identify the contributing factors, for which the five whys exercise can be used. Alternatively, the factors can be related to common cause areas, such as people, procedure, policies, place, systems, suppliers, skills and surroundings. As the categories are written on the diagram, the team needs to assign and write the causes of the effect under the appropriate category; they then define actions to treat, terminate, tolerate or transfer the cause.

8.5.1.5 Plus/delta

This feedback technique, which identifies positives as well as the negatives of an event, gives team members the opportunity to discuss the practices, ideas and patterns that have a positive effect on their collaboration and delivery, as well as reflect on those elements that require improvement or adjustment.

8.6 EMERGENT DOCUMENTATION

In an Agile delivery the focus is on producing only the relevant documentation in line with the emergence of the system. Documentation should only be produced if and when it adds value, and should always be fit for purpose, i.e. suitable for the audience.

It can be tempting to miss out documentation tasks as it is often not the most exciting part of the job, so teams need to find a way to make sure documentation is kept up to date. The easiest way to do this is to specifically make documentation production a standard part of the definition of ‘done’ (see Section 10.2.1) or have specific stories in place to create appropriate documents, or add documents as acceptance criteria on some stories. This means that before any piece of work can be signed off as complete, the necessary documentation has to be completed as well.

Typical types of IT systems documentation include:

- design documentation;
- code documentation;
- test documentation;
- business user documentation;
- operational documentation.

There are some occasions when it is necessary to produce documentation in advance of product development. For example, sometimes there may be a regulatory or commercial reason to produce a document or a specification for an interface that will enable an external team to develop something that will interact

with the product that is being developed.

Design documentation

In Agile there is generally no specific guidance on what design documentation should be produced. Generally the team will inspect and adapt to produce whatever design documentation adds value. However, in the majority of cases there will need to be some design documentation to ensure that the product can be effectively supported and maintained throughout its lifetime.

A key concept in Agile is emergent or opportunistic design, meaning that the design evolves as the team learn via the inspect and adapt cycles. The implication for design documentation therefore is to not try to define everything up front as this will restrict the ability to implement design in an emergent way.

This is easy when there are only one or two teams working on a product. However, once a product has multiple teams working on it there is typically a need to ensure that all teams are working towards the same design pattern. In this case high-level foundation design principles may need to be documented. The principles should not be a complete product design document; rather they should contain enough detail for the teams to work on for the next couple of iterations/sprints (or whatever time period is appropriate). This is called EDUF (enough design up front – (DSDM Consortium, 2014b)). It focuses the teams to work towards an intentional design, yet avoids wasting effort creating a fully completed design document up front.

Code as documentation

Code written by a team should be self-documenting, meaning it should be written in a way that is easy to understand by anyone, making extra documentation unnecessary. This can be achieved either by adding clear comments within the code or by ensuring the code is so simple and clear that its intent is obvious.

Focusing on simplicity of code aligns to the software craftsmanship movement (Software Craftsmanship, 2009), which states that a software craftsman will create self-documenting, simple-to-understand code that is easy to maintain in the future. If code does need explaining then this is a ‘code smell’ (Fowler et al., 1999) and requires refactoring (see Section 8.10.1).

Test documentation

Test documentation (e.g., test strategy, test scripts and so on) should be produced to support whatever testing is being implemented, (see Section 7.4).

Test documentation can also be used to provide a specification of the system (for examples see ‘specification by example’ and ‘Behaviour Driven Development’ (Section 7.4.1).

Business user documentation

Business user documentation is there to aid users of a system to use the system effectively. It may be online help, supporting documentation or whatever is fit for purpose. Some IT products do not require business user documentation, as the interface is specifically designed to be very easy to use. Other technology products

do require business user documentation and where this is the case the documentation is core to the delivery. There is no specific guidance about what business documentation should be produced in an Agile delivery; instead, the focus is on developing fit-for-purpose business documentation as the product emerges.

Operational documentation

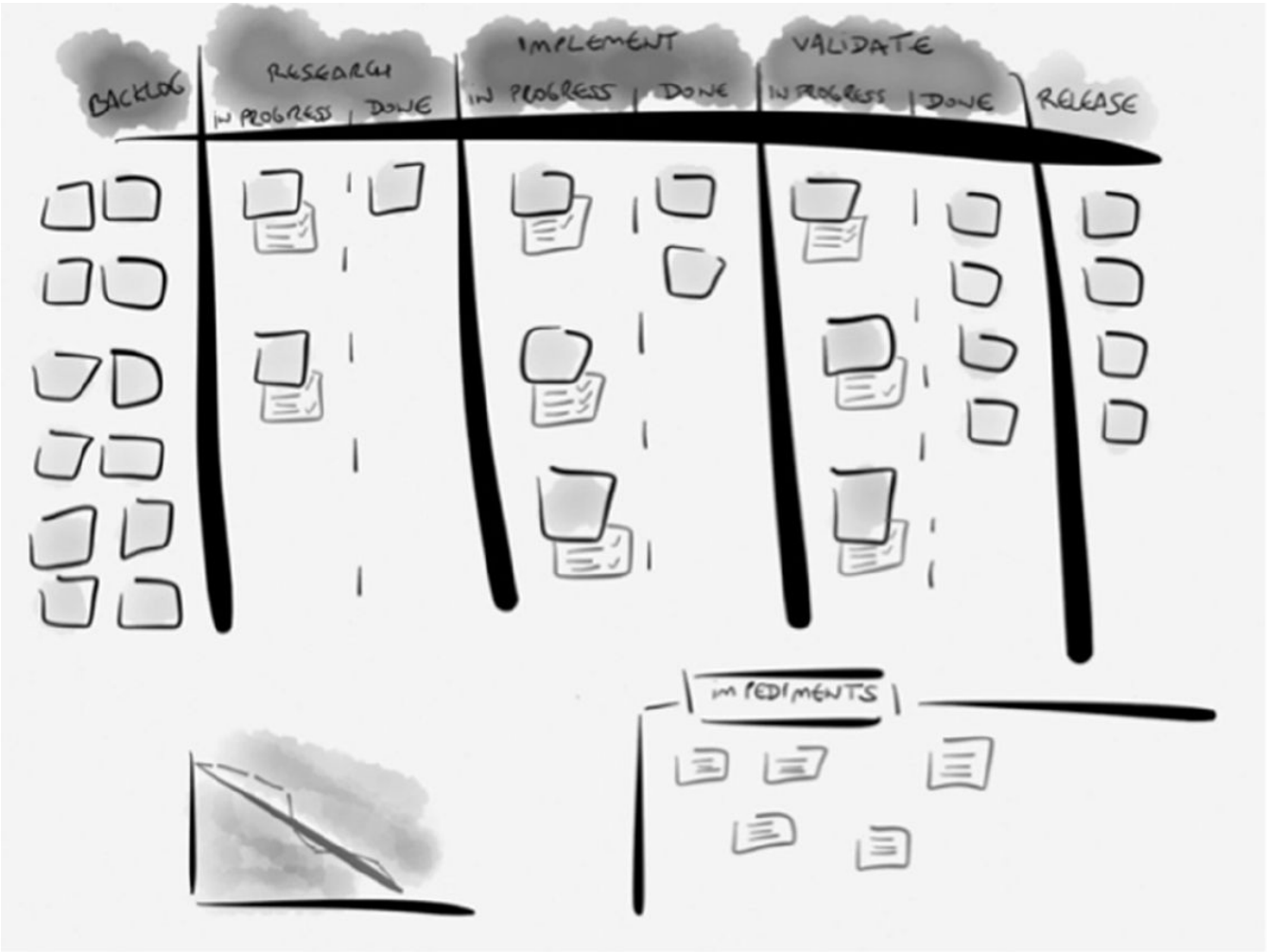
If a system is put into an environment that is controlled by a separate operational team, it is highly likely that documentation to describe the system to a level that the support team can support it will be required. Appropriate documentation needs to be inspected and adapted based on the operational team's requirements. Therefore it is essential that the operational team is considered a key stakeholder and that stories are created that represent their requirements.

8.7 VISUAL BOARDS

8.7.1 Information radiator

An 'information radiator', as shown in Figure 8.5, is intended to be openly visible and available, 'radiating' information to everyone who sees it. The use of information radiators is a great way of conveying information about the current state of a delivery.

Figure 8.5 Information radiator



Information radiators can be either physical, for example on a wall, whiteboard or

similar, or virtual, for example, as part of an Agile software planning tool. Many people still prefer to use physical boards as it allows direct interaction; for example, a team member can pick up a story or task card that represents what they are working on and physically move/progress it once the work is complete. This generally offers much more satisfaction than just changing a virtual ticket status. It is also immediately visible to everyone that progress has been made, without the need to access a piece of software.

However, there are limitations to the use of a physical board. For example, if a team is distributed across different countries, offices or even different rooms, team members may not be able to see and interact with a physical board in a single location. In such cases, visual boards held in the virtual world can be extremely useful; however, Agile teams sometimes refer to these as ‘information refrigerators’ as there’s the risk that no one ever keeps them up to date.

Information radiators are a key source of information for daily stand-up meetings, show and tells and retrospectives. For example, at a daily stand-up team members can talk about the progress made on stories and tasks represented on the board. For a show and tell, visual boards provide information on completed stories and tasks. For a retrospective, boards can show information that might need attention, for example tasks that have been blocked, or tasks that have been on the board for a long time.

As a minimum any information radiator should show the current state of tasks/stories a team are working on and how far they have to go to get to ‘done’ status. However, it can also include other information, for example, who is currently working on what task(s).

It can also show a burn-down chart, a burn-up diagram, blockers list or project information such as the key items from the risks assumptions issues and dependencies (RAID) log.

> The only person who can move a story to complete status (meaning the acceptance criteria are now at ‘done’ status) on a visual board is the customer.

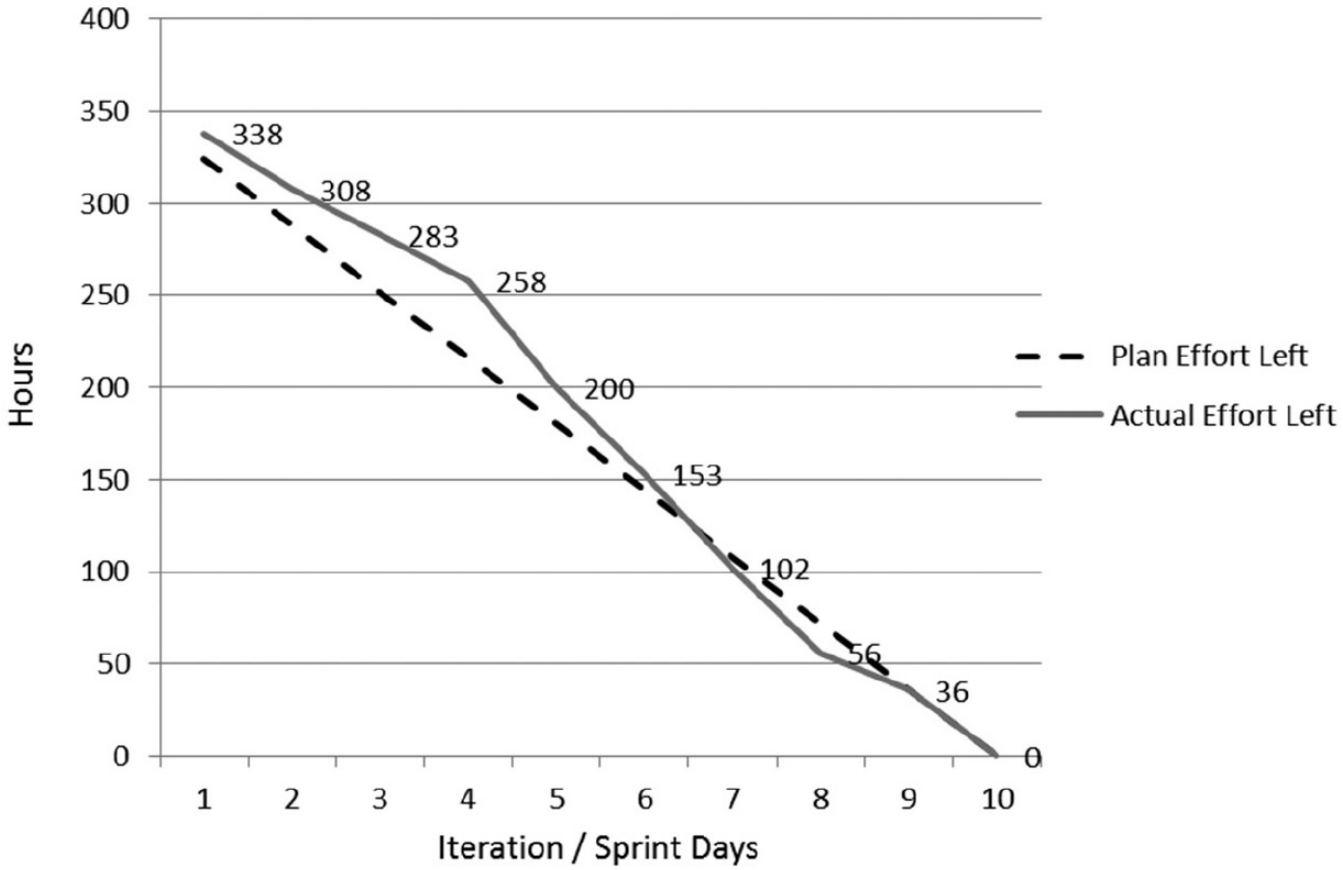
8.7.2 Burn-down chart

A burn-down chart normally compares planned effort left to complete a task against the actual effort left. It works well for a team who have estimated the effort required to complete tasks within an iteration/sprint in hours. Initially the chart would show the predicted rate at which a delivery will occur by forecasting how much effort (in hours) should be left on a particular day within the sprint/iteration. This is plotted as a line on the chart. On a daily basis each team member then updates how much effort they actually have left against the tasks they are delivering, and the total of the latest estimated hours left is then plotted onto the chart each day. The two lines should be broadly following each other through the iteration/sprint.

The following example relates to a single iteration/sprint where it would be usual to use hours as a measure of effort.

A burn-down chart (Figure 8.6) can be used to predict whether a team will succeed in delivering everything within the iteration/sprint by extrapolating the ‘actual effort left’ line based on latest forecasts. When properly planned, the chart should show that all the work will be done just before the end of the sprint/iteration.

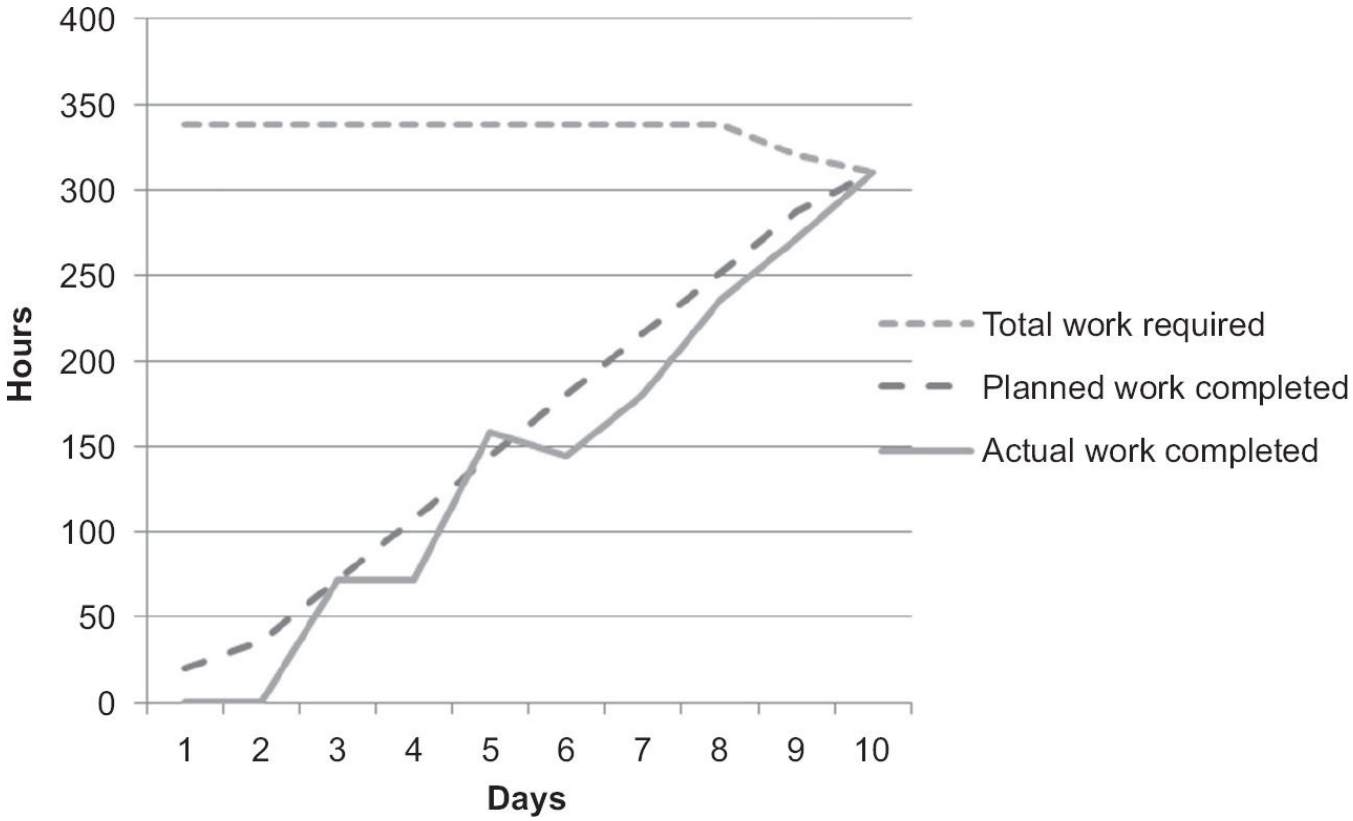
Figure 8.6 Burn-down chart - example 1



Story completion (measured in story points) can also be used for iteration/sprint burn-down; however, this is likely to create a chart that is not granular enough for effective measurement of progress. Release or project burn-down charts are also typically created and in those cases forecast and actual story point burn-downs would be used.

A burn-up chart (Figure 8.7) monitors the total work required, normally in hours or story points, and then displays the actual work completed; where the two lines intersect the iteration/sprint is ‘done’.

Figure 8.7 Burn-up chart - example 2



Typically a ‘planned work completed’ line is also added to the burn-up chart.

8.7.4 RAID log

A RAID log is a single repository of all key information about a delivery that is not expressed on the backlog or in other documentation. ‘RAID’ stands for:

- Risks.
- Assumptions.
- Issues/actions.
- Decisions (or dependencies).

Displaying the major RAID log items on the visual board can be very useful to provide a quick, visual overview of risks and so on, though it is important not to ‘overload’ the board with information. The best approach is to only flag RAID items that are, or could be, impacting on what the team is doing within the current sprint/iteration or release.

8.8 SUSTAINABLE PACE

Sustainable pace is one of the practices that was introduced into Agile by eXtreme Programming (XP) (see Section 14.1). eXtreme Programming advocates frequent ‘releases’ in short development cycles. This is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

In the early 1900s, Ford Motor Company ran dozens of tests to discover the optimum work hours for worker productivity. They discovered that the ‘sweet spot’ is 40 hours a week and that, while adding another 20 hours provides a minor increase in productivity, this increase only lasts for 3 to 4 weeks, and then turns

negative (James, 2012).

Research by the Roundtable (Holloway Consulting, 2014) in the 1980s found that you could get short-term gains by going to 60- or 70-hour weeks very briefly – for example, pushing extra hard for a few weeks to meet a critical production deadline. However, increasing a team's hours in the office by 50 per cent (from 40 to 60 hours) does not result in 50 per cent more output. In fact, the numbers may typically be something closer to 25–30 per cent more work in 50 per cent more time.

This is because by the eighth hour of the day, people's best work is usually already behind them (typically turned in between hours two and six). In hour nine, as fatigue sets in, they're only going to deliver a fraction of their usual capacity. And with every extra hour beyond that, the workers' productivity levels continue to drop, until at around 10 or 12 hours they hit full exhaustion.

Overtime is only effective over very short periods of time. This is because daily productivity starts falling off in the second week, and declines rapidly with every successive week as burnout sets in (Chapman, 1909).

Burnout is a psychological term that refers to long-term exhaustion and diminished interest in work. Tired teams will make more mistakes, produce more defects, deliver less and with a reduced quality level. Indeed, burnout could ultimately lead to a team member deciding to leave the organisation.

Additionally working a lot of overtime creates a level of burnout that sets in far sooner, is far more acute, and requires much more to fix than most managers or workers think it does.

> Working overtime sucks the spirit and motivation out of your team. When your team becomes tired and demoralized they will get less work done, not more, no matter how many hours are worked. Becoming over-worked today steals development progress from the future.
>
> (Wells, 2009)

8.9 FOCUS ON QUALITY

This section looks at the concept of quality in an Agile context and gives an overview of the main quality focus practices.

8.9.1 Functional quality

Ensuring functional quality is about delivering the features and functionality the customer wants. Lack of functional quality, in other words not delivering the system the customer wants, is a common criticism levelled at IT deliveries.

Customer collaboration, one of the four Agile Manifesto statements is a key focus of ensuring functional quality. It feeds into defining acceptance criteria for each feature, the review of stories throughout development and at the end of development, as well as into the definition of 'done' (see Section 10.2).

8.9.2 Technical quality

There are some practices used specifically in software development that are aimed at ensuring that software is of an appropriate level of technical quality. The ones listed below are the foundation technical quality practices that are expected to be in place in any Agile team, and will be covered in more detail in Section 8.10. For a description of other more technically biased practices that may be implemented see eXtreme Programming (Section 14.1).

- **Refactoring** – this is changing the design of a system without changing its behaviour.
- **Continuous integration** – this is about continuously ensuring that everything works together in an integrated way.
- **Test Driven Development** – Test Driven Development (TDD) is a practice where test cases are written before the actual functionality is developed. Tests are written for each unit/component of code.

8.10 MAJOR AGILE TECHNICAL PRACTICES

This section provides a foundation-level understanding of some of the major Agile technical practices. A more exhaustive list of technical practices may be found in the description of eXtreme Programming in Section 14.1.

8.10.1 Refactoring

Refactoring (Fowler et al., 1999) is changing the design of a system without changing its behaviour. It is used to create a design and architecture for a product that is fit for purpose and doesn't contain technical debt (see Section 10.4.1).

Teams need to have confidence that, as they develop the system design (see Section 9.2), they can revisit the design and architecture and make changes based on learning. Refactoring is specifically focused on improving and ensuring continual quality of the design of the system (including the database), and should be done as a matter of normal development practice to prevent the software becoming laden with technical debt. Continual attention to refactoring ensures that the code is of good design and easily maintainable. Code that is easy to maintain will produce fewer defects and ensure that systems are easy to support and maintain.

One way to implement refactoring is Test Driven Development (TDD – see Section 7.4.1.2).

An analogy to explain refactoring is that, when camping, it is good etiquette to leave the campsite in as good or better condition than when you arrived. The same applies to software development: when a developer designs and implements a new feature, it is good practice to make sure the surrounding code, tables, documentation and so on are still in good shape.

When refactoring, it is important to have a suite of tests which should be executed before anything is refactored to ensure they are running correctly. After refactoring has been completed the tests should be run again to ensure there has been no

adverse effect. This ability is closely aligned with the concept of continuous integration.

8.10.2 Continuous integration and automated testing

Not integrating testing throughout the software development lifecycle poses a very significant risk as it unnecessarily extends feedback loops (see Section 8.1). A nightmare scenario is that important testing is only performed at the end of a release or project, and if at that point a significant problem is found nothing can be done about it without extending timescale and costs.

Therefore Agile deliveries implement continuous integration (Fowler, n.d.), which means that any change to a product will initiate full regression tests of the whole product (i.e. tests to ensure that adding something has not caused the overall software environment to break).

Continuous integration enables team members to frequently and independently integrate their work with the core product and therefore ensure that the overall product continues to work in an integrated manner. Typically team members will integrate their work multiple times a day, and at the very least on a daily basis. The integrated work is continually verified by build and test software tools. This gives the team confidence that the system works in an integrated way in any environment (e.g. an integration test environment), and that they will not hit a significant blocker late in the development cycle by discovering an integration problem.

There are a number of principles associated with continuous integration:

- Maintain a central source control system.
- Automate the build (see Section 8.10.3).
- Make the build self-testing – automated regression tests.
- Everyone commits to the code baseline every day.
- The code baseline is rebuilt every day (at least).
- Builds must be fast – especially if building multiple times a day.
- Test in environments that mirror the live environments wherever possible.
- Everyone can easily see the latest product (build).
- Everyone can easily see the results of builds – the focus of the team is to fix broken builds as soon as possible.
- Automate deployment where possible.

Teams should always be working on the latest version of a product. They should regularly pull the latest changes from the central source control system and then check-in their updated code. New code check-ins are detected by the continuous integration system. The system will then run a process to rebuild the changed software environment, and run unit and integration tests to see if anything has regressed. If there are any tests that fail, then the build fails and the developer who checked-in the code will be notified by the system. It is then the responsibility of

the developer who changed the code to resolve the problem on behalf of the team.

8.10.3 Automated builds

Continuous integration cannot be performed without automated testing and build tools, because it is continuous and the overhead of continuous testing cannot be performed manually. Build automation is the creation of scripts that automatically perform developer tasks such as:

- compiling code;
- running tests;
- performing code analysis;
- assembling code components into features (also known as ‘build’);
- deploying to environments;
- creating system documentation.

Agile deliveries should have the ability to run automated build continuously and therefore deploy continuously. In the event of a broken build, priority should be given to fix the build so that the team can continue their work with confidence. If a build has been successful, it is then possible to automate the deployment to a test environment and run the automated continuous integration tests. Automated builds should be scheduled to run on a regular frequency, at least daily.

8.10.4 Code review and peer review

Whilst automated builds are a good way to automatically check many aspects of software, they cannot detect everything. Humans are arguably much better than machines at checking that code is well written, designed and maintainable. There are a number of different ways to approach this: two common Agile approaches are that either a developer can submit completed code (for example) for peer review before committing the code, or the XP practice of pair programming can be adopted (see Section 14.1).

Peer reviews are performed with people of a similar level of experience to the person that created the product to be reviewed. The key is that peer reviews must not slow down the Agile process; therefore they may only be focused on key products or may be implemented more as a sample test intermittently. As usual in Agile, common sense should prevail and the amount and rigour of code review should be implemented in line with the complexity and rigour of the product required.